+---------------------------+

| COE127L |

| PROJECT: VIRTUAL MEMORY |

| DESIGN DOCUMENT |

+---------------------------+

---- GROUP ----

>> Fill in the names

Cortez, Rex S.

Roque, Derik Robert L.

Masigla, Vernon Louise

PAGE TABLE MANAGEMENT

=====================

---- DATA STRUCTURES ----

>> A1: Copy here the declaration of each new or changed `struct' or

>> `struct' member, global or static variable, `typedef', or

>> enumeration. Identify the purpose of each in 25 words or less.

struct thread

{

...

#ifdef VM

struct hash s\_page\_table; /\* Supplemental page table for process \*/

struct lock s\_page\_lock; /\* Lock for page table \*/

void \*saved\_esp;

bool syscall\_context;

struct list mmap\_list;

int next\_mmap;

#endif

...

}

struct s\_page\_entry

{

enum entry\_type type; /\* Type of entry \*/

uint8\_t \*uaddr; /\* User page address (page-aligned) \*/

bool writable; /\* Whether page is writable \*/

union

{

struct file\_based file;

struct memory\_based memory;

} info; /\* Attributes of entry \*/

struct frame\_entry \*frame; /\* Frame entry if frame is allocated \*/

struct hash\_elem elem; /\* Entry in thread's hash table \*/

struct lock l; /\* Lock for when this page is "in play" \*/

};

This struct represents a supplemental page table entry for a process. The

fields are documented above. It can represent a file- or memory-based page

based on the union aspect, whose structs are defined below.

enum entry\_type

{

FILE\_BASED,

MEMORY\_BASED

};

These are the valid types for the type of supplemental page table.

struct file\_based

{

struct file \*f; /\* File struct to access the filesystem \*/

off\_t offset; /\* Current offset into the file \*/

size\_t zero\_bytes; /\* Number of zero-padding in this page \*/

bool init\_only; /\* Marks a page as write-to-swap (i.e. .bss) \*/

};

Defines the data needed for a file-based supplemental page table entry. If

init\_only is true, the page will be read from disk but once modified

transformed into a memory\_based page, i.e. one that can be swapped.

struct memory\_based

{

bool used; /\* Has this page been swapped before \*/

bool swapped; /\* Is this block swapped \*/

block\_sector\_t swap\_begin; /\* The starting swap block containing the page\*/

};

Defines the data needed for a memory-based supplemental page entry. The

used flag is to ensure that if page has never been used we will not bother

swapping it.

enum vm\_flags

{

VM\_ZERO = PAL\_ZERO /\* Zero page contents. \*/

};

This shadows the flag in palloc.h to mark a page as zero-initialized.

static struct list\_elem \*clock\_hand; /\* The hand of the clock algorithm \*/

static struct list frames; /\* List of frame\_entry for active frames \*/

static struct lock frames\_lock; /\* Protects struct list frames \*/

The above globals for frame table management are described by their comments.

struct frame\_entry

{

struct thread \*t; /\* Owner thread \*/

struct s\_page\_entry \*spe; /\* Owner page entry \*/

uint8\_t \*kaddr; /\* Physical address \*/

struct list\_elem elem; /\* Linked list of frame entries \*/

bool pinned; /\* Whether this frame is pinned or not \*/

};

Defines the data needed for a frame table entry. The comments describe the

various elements.

#define BLOCKS\_PER\_PAGE PGSIZE/BLOCK\_SECTOR\_SIZE

struct bitmap \*swap\_table; /\* Directory of free/used swap blocks \*/

struct lock swap\_lock; /\* Protects swap\_table \*/

The above globals and definitions are for swap block management.

---- ALGORITHMS ----

>> A2: In a few paragraphs, describe your code for locating the frame,

>> if any, that contains the data of a given page.

We assume this question begins at the point where a page fault is

generated, that is, the frame containing the data for a given page is not

resident in memory and mapped for the process generating the request.

Beginning in the page fault handler, if the not\_present flag is set, we

call into our process page loader, page\_load(). This function changes the

fault address into a base page address and looks it up in the process's

hash map of supplemental page table entries, which is keyed on user

virtual page address. If no entry is found this is an invalid access and

we return failure.

If an entry is found, we read whether it is a file-based or memory-based

page and dispatch appropriately. First we use our frame allocation system

to fetch a frame for use, which may result in an eviction if no frames are

free. If the page is file-based we read the frame contents back from the

disk. If the page is memory-based, we read the frame back from swap if it

was swapped, or initialize it to zero if it was never used (and therefore

swapped) before.

Once the frame is repopulated it is mapped back into the process's page

table. The page\_load() function then returns success to the page\_fault

handler, which returns and allows the process to resume.

>> A3: How does your code coordinate accessed and dirty bits between

>> kernel and user virtual addresses that alias a single frame, or

>> alternatively how do you avoid the issue?

We avoid the issue by always acccessing data using user virtual

addresses. The only time the kernel addresses are used are to populate

data into the frame when handling a page fault, which does not count as

accessed or dirty anyway.

---- SYNCHRONIZATION ----

>> A4: When two user processes both need a new frame at the same time,

>> how are races avoided?

We avoid races with two main mechanisms. First, the frame table is

protected by a lock, so only one process may access and modify it at a

time. The functions in palloc.h are also protected by locks to prevent

races.

Next, once a frame has been selected it is marked as pinned, which

excludes it from consideration by other processes while the eviction and

loading operations are done to it.

We take care to only select and pin a frame within the critical section so

that processes can proceed with evicting and loading frames concurrently.

---- RATIONALE ----

>> A5: Why did you choose the data structure(s) that you did for

>> representing virtual-to-physical mappings?

We used a hash map because it allows for an O(1) and space-efficient

mechanism for managing the mapped pages of each process. We knew that we

would need to support fast lookups by user virtual addresses, but kernel

(physical) addresses would only be needed when installing frames.

It seemed fairly logical to have one data structure to represent a frame

in the frame table, which is primarily concerned with tracking the

physical-to-virtual mappings, pinning, and selection by the clock

algorithm.

We were satisfied with our union data structure to manage a supplemental

page entry. It made things relatively simple and actually turned out to be

a boon when we realized that .bss segments would need to start as

file-based entries but transform into memory-based entries.

PAGING TO AND FROM DISK

=======================

---- DATA STRUCTURES ----

>> B1: Copy here the declaration of each new or changed `struct' or

>> `struct' member, global or static variable, `typedef', or

>> enumeration. Identify the purpose of each in 25 words or less.

The relevant data structures are documented above in section A.

---- ALGORITHMS ----

>> B2: When a frame is required but none is free, some frame must be

>> evicted. Describe your code for choosing a frame to evict.

To perform frame eviction we implemented the Clock Algorithm, which

approximates LRU. As described in Question A4 we protect the frame list

with a lock to address concurrency issues.

The algorithm begins by advancing the clock hand to the next unpinned

frame in the circularly-linked list. It is important to ensure that the

frame pointed to by the clock hand is unpinned to set it up for a

straightforward invocation of the Clock Algorithm.

Then it runs the Clock Algorithm as follows: for each frame pointed to by

the hand, if it is a pinned frame, ignore it. Else if its accessed bit is

on, turn it off. Else, pin and return the frame. Then advance the clock

hand.

If the hand has wrapped around to where it started, pin and return that

frame. This \*should\* happen automatically with the access bit

modification, though it is possible that the frame was accessed by another

thread concurrently.

>> B3: When a process P obtains a frame that was previously used by a

>> process Q, how do you adjust the page table (and any other data

>> structures) to reflect the frame Q no longer has?

When P obtains a frame that was used by Q, we first pin the frame, acquire

the lock for the supplemental page table entry associated with that page,

and then remove it from process Q's page table. This means that process Q

will fault upon any accesses to this frame for now on, but it will have to

block on acquiring the supplemental page table entry lock before

un-evicting its frame (to avoid contention with the swapping occurring on

P's thread).

From here, depending on whether Q's frame was file-based on memory based

it will be written to disk or swap as appropriate and then the

supplemental page table entry will be updated to reflect this. The primary

mechanism for knowing that the frame is now unmapped is that it is no

longer in the process's page table.

>> B4: Explain your heuristic for deciding whether a page fault for an

>> invalid virtual address should cause the stack to be extended into

>> the page that faulted.

First, we try loading the page containing the fault address into memory,

in case we just needed to swap it in. If that fails, then we check for

stack growth conditions.

First, we check if the faults were due to a PUSH or PUSHA instruction, in

which case we fault 4 and 32, respectively, below the stack pointer.

We extend the stack when we detect PUSH and PUSHA accesses.

Next, we check to see if we faulted on a virtual address above the stack

pointer. Any faults above the stack pointer (and in user memory space)

trigger stack growth.

---- SYNCHRONIZATION ----

>> B5: Explain the basics of your VM synchronization design. In

>> particular, explain how it prevents deadlock. (Refer to the

>> textbook for an explanation of the necessary conditions for

>> deadlock.)

Our VM synchronization design has locks at the following layers:

The frame table has a single lock to protect its access. We avoid deadlock

by ensuring that there is not a circular dependency between any code that

acquires the lock, and the only lock that is acquired while it is held,

the supplemental page table entry lock for the supplemental page table

entry of concern.

The swap table bitmap that tracks used and free bits. No locks are

acquired while it is held, so there is no risk of a circular dependency

and deadlock.

Each process has a lock protecting its hash table of supplemental page

table entries. Its use is constrained to looking up entries during a page

fault or modifying the data structure itself during insertions or

removals. The only potential issue is that the lock on the supplemental

page table entry must be acquired before the hash table lock is released

during a lookup. This could have a potential for deadlock if one thread

held the page entry lock and attempted to do a lookup while the other did

the opposite (creating a cycle), but we eliminated any code that did this

by adding a pointer from a frame table entry to its associated page table

entry.

>> B6: A page fault in process P can cause another process Q's frame

>> to be evicted. How do you ensure that Q cannot access or modify

>> the page during the eviction process? How do you avoid a race

>> between P evicting Q's frame and Q faulting the page back in?

We have a lock on the supplemental page table entry, which must be

acquired before a page is evicted or unevicted. Therefore, if Q faults it

will block until P is done evicting its frame.

We prevent Q from being able to write the page mid-eviction (and thus

causing it to fault immediately) by removing the mapping from Q's page

table before beginning the eviction process.

>> B7: Suppose a page fault in process P causes a page to be read from

>> the file system or swap. How do you ensure that a second process Q

>> cannot interfere by e.g. attempting to evict the frame while it is

>> still being read in?

We prevented this by implemented pinning for frames. Whenever a frame is

retrieved by a process for loading it is marked as pinned. Pinned frames

cannot be considered by the clock algorithm and therefore are not

candidates to be given to another process Q until they are unpinned, which

occurs after the loading process finishes.

>> B8: Explain how you handle access to paged-out pages that occur

>> during system calls. Do you use page faults to bring in pages (as

>> in user programs), or do you have a mechanism for "locking" frames

>> into physical memory, or do you use some other design? How do you

>> gracefully handle attempted accesses to invalid virtual addresses?

Our implementation uses page faults to bring in pages as usual, since all

of our kernel code accesses user data using the user virtual addresses. In

project 2 we elected to go with the eax-esp trick for checking user data

before access, and so this naturally extended to the page faulting

code.

Specifically, in our page fault handler we first attempt to pull in

the page. If that fails (i.e. the page was not valid) then we see if we

need to extend the stack. Then, if the processor is in the user context we

kill the process, but if it is in the kernel process we check for the

special variables used to signal an error to the kernel.

---- RATIONALE ----

>> B9: A single lock for the whole VM system would make

>> synchronization easy, but limit parallelism. On the other hand,

>> using many locks complicates synchronization and raises the

>> possibility for deadlock but allows for high parallelism. Explain

>> where your design falls along this continuum and why you chose to

>> design it this way.

For simplicity we began by using as coarse locks as possible. For this

reason we only have one lock for the frame table, one lock for the swap

allocation bitmap, and one lock for each process's supplemental page table

hashmap. Still, we attempted to make the critical sections as small as

possible by only holding these locks when accessing the data structures

they protect and never holding them recursively.

This worked well and this implementation did not appear to have any

deadlocks. However, we soon found that we needed to add locks to the

supplemental page table entries to prevent the race conditions discussed

above, specifically when two processes are attempting to evict and unevict

the same frame. While the finer-grained locks solved this issue while

still allowing for the same degree of parallelism, the potential for

deadlock was introduced. We solved it by being as careful as possible with

our code, reasoning about possible concurrent access patterns and

attempting to eliminate any potential circular lock dependencies.

In summary, our design approach was to have as coarse locks as possible

while maximizing concurrency, only introducing fine-grained locks when

needed while attempting to keep their uses orthogonal to other locks to

keep potential deadlock conditions to a minimum. Though we discovered and

avoided many deadlocks, it is admittedly very hard to say with confidence

that we avoided them all.